As with previous chapters, we are going to lay out the learning objectives for this chapter, and split them into what you should get on the first reading vs the second reading.

1 First Reading: Understand Key Ideas

On your first reading, the goal is to get the big ideas, and build the mental scaffolding in which to hold the details on your second reading—the learning objectives for your first reading are generally **Remember** and **Understand** objectives. If you are missing a few details (but not completely lost), it is ok to move ahead, and come back to them the next time through—you might ask questions on the course forums too! Note you should **sleep at least one night between the first and second reading** to give your brain time to process the information.

- **Recursion: Intro**
  - **Define** *recursion*.
  - **Explain** why you should master recursion.
- **Reading Recursive Code**
  - **Explain** how to execute recursive functions by hand.
  - **Explain** why executing recursive functions by hand does not need any new skills.
  - Note: even though we typically suggest carefully working through the examples/videos on the second reading, you should be able to follow and completely understand Video 7.1 as it is using C semantics you have been working with since Chapter 2.
- **Writing Recursive Code**
  - **Explain** how The Seven Steps can result in code that is naturally recursive.
  - **Explain** why a recursive function that does nothing but call itself (such as `factorial_broken`) does not work.
  - **Define** *base case*.
  - **Define** *recursive case*.
  - **Explain** what it means for a recusive function to "make progress towards the base case."
  - **Explain** why a recursive function might need more than one base case.
  - **Explain** why some people think "recursion is slow."
  - **Define** *memoization*.
  - Note: *dynamic programming* is basically just memoization. We don't cover dynamic programming here, but if you can write a recursive algorithm that works but duplicates work, then you can just memoize it, and you have a dynamic programming solution. An Algorithms class should cover dynamic programming.
- **Tail Recursion**
  - **Define** *head recursion*.
  - **Define** *tail recursion*.

  - **Define** *tail call.*
  - **Explain** how to tell if a function is head or tail recursive.
  - **Recognize** that tail recursion can be optimized to eliminate frame creation.
    * Note: on the first reading, we want you to just recognize this can be done. Don't get hung up on the details until the second reading, after you are more familiar with the material.
  - **Recognize** that tail recursion can be converted to iteration and vice-versa.
    * Note: on the first reading, we want you to just recognize this can be done. Don't get hung up on the details until the second reading, after you are more familiar with the material.

- **Functional Programming**
  - **Define** *functional programming language.*
  - **Explain** why it might be beneficial to try functional programming.

- **Mutual Recursion**
  - **Define** *mutual recursion.*
  - **Recall** (from before) the definition of a *function prototype*
  - **Define** *recursive descent parsing.*

- **Theory**
  - **Define** *induction* (the mathematical proof technique).
  - **Explain** the relationship between induction and recursion.
  - Note: we do not go into formally proving code here. However, if you take an Algorithms class, you should do a lot of proofs by induction, and get more in-depth on this relationship between induction and recursion. Remember that a proof is worth a thousand test cases.
  - **Define** *well ordering.*
    * Note: AoP assumes you already know what a *total ordering* is from your prior math classes. If not, you should **define** *total ordering* first.
  - Note: defining/understanding *ordinal* numbers is NOT a learning objective of this section. We reference them for mathematical accuracy, but if you are not familiar with them from your prior math classes, you can just ignore that part of the discussion and think about the natural numbers.
  - **Explain** how the "less than" relationship on the well ordering of the input type can ensure that a recursive function terminates (does not recurse infinitely).
  - **Define** *measure function.*
  - **Explain** how a measure function can ensure termination of a recursive function.

2 Second Reading

As always, we strongly recommend you sleep between your first and second reading. The second reading is where you want to focus on the higher-level learning objectives, which build on the lower-level learning objectives you worked on in the first reading. Sleeping gives your brain a chance to process the information from the first reading.

- **Reading Recursive Code**
  - **Execute** recursive code by hand.
- **Writing Recursive Functions**
  - Note: for now, these learning objectives are restricted to recursion on natural numbers (unsigned ints). Much later, you will recurse on other types.
  - **Assess** when recursion is a good approach for an algorithm.
  - **Write** recursive algorithms
  - **Translate** recursive algorithms into code.
  - **Assess** how many base cases are needed for your recursive algorithm/code.
  - **Assess** when your recursive algorithm duplicates work (making the same recursive call many times).
  - Note: as mentioned earlier, we don't do dynamic programming here, so we don't have a learning objective to fix the duplicated work with dynamic programming.
- **Tail Recursion**
  - **Determine** if a recursive function is head or tail recursive (or a mix of the two).
  - **Execute** tail recursive code by hand, without performing tail call optimization.
  - **Explain** tail call optimization.
  - **Execute** tail recursive code by hand, performing tail call optimization.
  - **Rewrite** tail recursive code as iteration.
  - **Rewrite** iterative code as tail recursion.
- **Functional Programming**
  - None for second reading: we just want you to know what functional programming is, and that it is good to try sometime.
- **Mutual Recursion**
  - **Execute** mutually recursive code by hand.
  - **Assess** when a step in your algorithm should be abstracted out to a function (Note: you have seen this LO before)
  - **Assess** when a step in your algorithm translates into a function you have already written.
  - **Write** mutually recursive functions.

  - **Declare** function prototypes where needed.
  - Note: there are not any more learning objectives around recursive descent parsing—it is a good thing to know how to do, but not covered here in any more detail than as motivation for mutually recursive functions.

- **Theory**
  - None for second reading. There are a LOT of (big, complex) higher level learning objectives that are great to master on these topics, but they are outside the scope of this class. If you are taking an Algorithms class, you should go into more depth on this material there.

3 Key Learning Objectives

The most key learning objectives from this Chapter are:

- **Execute** recursive code by hand.
- **Assess** if recursion is a good approach to solving your problem.
- **Write** recursive algorithms and code.